

## DESIGN AND ANALYSIS OF ALGORITHMS LABORATORY

Course Code : 20ISL56

L: T:P: 0:0:2

Exam Hours : 3

Credits : 2

CIE Marks : 25

SEE Marks : 25

### PART A

#### PROGRAM 1:

##### PROBLEM STATEMENT:

Implement and analyze Quick Sort algorithm.

##### CONCEPT & PROCEDURE

Quick sort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heap sort.

Quick sort is a divide and conquer algorithm. Quick sort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quick sort can then recursively sort the sub-arrays.

The procedure is:

1. Pick an element, called a *pivot*, from the array.
2. *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Average performance of quick sort is  $O(n \log n)$

```
#include<stdio.h>
#include<time.h>

int partition(int a[], int low, int high)
{
    int pivot = a[low], i=low, j= high+1;
    int temp;
    while(i<j)
```

```

    {
        do
        {
            i++;
        }while(pivot>=a[i] && i<high);
        do
        {
            j--;
        }while(pivot<a[j]);
        if(i<j)
        {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
    a[low] = a[j];
    a[j] = pivot;
    return j;
}

void quick_sort(int a[], int low, int high)
{
    int s;
    if(low<high)
    {
        s = partition(a,low,high);
        quick_sort(a,low,s-1);
        quick_sort(a,s+1,high);
    }
}

int main()
{
    int a[10000],n,low,high,i;
    clock_t st,end
    printf("Enter no. of elements");
    scanf("%d",&n);
    printf("Random numbers generated are");
    for(i=0;i<n;i++)
    {

        a[i]=rand()%100;
    }
}

```

```

        printf("%d\t",a[i]);

    }

    low=0;
    high=n-1;

    st=clock();

    quick_sort(a,low,high);

    end=clock();

    printf("Sorted Array :");
    for(i=0;i<n;i++)
    {
        printf("%d  ",a[i]);
    }

    printf("Time required to sort the given elements is %f",(end-st)/CLOCKS_PER_SEC);

    return 0;
}

```

### **OUTPUT:**

Enter the value of n

5

Random numbers generated are

46     30     82     90     56

Sorted array is

30     46     56     82     90

Time required to sort the given elements is 0.549451

## PROGRAM 2:

### PROBLEM STATEMENT:

**Implement and analyze Merge Sort algorithm**

### CONCEPT & PROCEDURE

Merge sort is a perfect example of a successful application of the divide-and-conquer technique.

- Split array A[1..n] in two and make copies of each half in arrays B[1.. n/2 ] and C[1..n/2]
- Sort arrays B and C
- Merge sorted arrays B and C into array A as follows:
  - a) Repeat the following until no elements remain in one of the arrays:
    - i) Compare the first elements in the remaining unprocessed portions of the arrays
    - ii) Copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - b) Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Average performance of merge sort is  $O(n \log n)$

### PROGRAM

```
#include<stdio.h>
#include<time.h>

int i,j,k,l,h,n,m;
int a[50], c[50];
clock_t st,end;
void merge(int a[], int l, int m, int h);
void merge_sort(int a[], int l, int h);
int main()
{
    printf("Enter length of array :");
    scanf("%d",&n);
    printf("Enter elements into the array : ");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }

    st=clock();
```

```

merge_sort(a,0,n-1);

end=clock();

printf("Time required to sort the given elements is %f",(end-st)/CLOCKS_PER_SEC);

printf("Sorted Array : \n");

for(i=0;i<n;i++)
{
    printf("%d",a[i]);
}
return 0;
}

void merge_sort(int a[], int l, int h)
{
    int m;
    if(l<h)
    {
        m = (l+h)/2;
        merge_sort(a,l,m);
        merge_sort(a,m+1,h);
        merge(a,l,m,h);
    }
}

void merge(int a[], int l, int m, int h)
{
    i=l, j=m+1, k=l;
    while(i<=m && j<=h)
    {
        if(a[i]<a[j])
        {
            c[k] = a[i];
            i++;
            k++;
        }
        else
        {
            c[k] = a[j];
            j++;
            k++;
        }
    }
}

```

```

        k++;
    }
}
while (i<=m)
{
    c[k++] = a[i++];
}
while (j<=h)
{
    c[k++] = a[j++];
}
for (i=1; i<=h; i++)
{
    a[i] = c[i];
}
}

```

### **OUTPUT:**

Enter the value of n

5

Random numbers generated are

46    30    82    90    56

After sorting:

30    46    56    82    90

Time required to sort the given elements is 0.000075190

### **PROGRAM 3:**

#### **PROBLEM STATEMENT:**

**Implement the following graph traversal techniques using decrease and conquer approach:**

- a. Breadth First Search method.**
- b. Depth First Search method.**

#### **Breadth First Search method.**

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbor nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbor nodes.

The BFS algorithm works as follows:

- Start by putting any one of the graph's vertices at the back of a queue.
- Take the front item of the queue and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
- Keep repeating steps 2 and 3 until the queue is empty.

### **Depth First Search method.**

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

The DFS algorithm works as follows:

- Start by putting any one of the graph's vertices on top of a stack.
- Take the top item of the stack and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
- Keep repeating steps 2 and 3 until the stack is empty.

### **BFS**

```
#include<stdio.h>
```

```
int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;
```

```
void bfs(int v)
```

```
{
```

```
for(i=1;i<=n;i++)
```

```
if(a[v][i] && !visited[i])
```

```
q[++r]=i;
```

```
if(f<=r)
```

```
{
```

```
visited[q[f]]=1;
```

```
bfs(q[f++]);
```

```
}
```

```
}
```

```
int main()
```

```
{
```

```

int v;
printf("\n Enter the number of vertices:");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
    q[i]=0;
    visited[i]=0;
}
printf("\n Enter the adjacency matrix form:\n");
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        scanf("%d",&a[i][j]);
printf("\n Enter the starting vertex:");
scanf("%d",&v);
bfs(v);
printf("\n The BFS Traversal is :\n");
for(i=1;i<=n;i++)
    if(visited[i])
        printf("%d\t",i);
    else
        printf("\n Bfs is not possible");

}

```

### **OUTPUT**

Enter the number of vertices: 5

Enter the adjacency matrix

0 1 1 1 0

0 0 0 0 1

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

Enter the starting vertex: 1

BFS Traversal is : 1 2 3 4 5

## DFS

```
#include<stdio.h>
int a[20][20],reach[20],n;
void dfs(int v)
{
    int i;
    reach[v]=1;
    for(i=1;i<=n;i++)
        if(a[v][i] && !reach[i])
        {
            printf("\n %d->%d",v,i);
            dfs(i);
        }
}
int main()
{
    int i,j,count=0;

    printf("\n Enter number of vertices:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        reach[i]=0;
        for(j=1;j<=n;j++)
            a[i][j]=0;
    }
    printf("\n Enter the adjacency matrix:\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&a[i][j]);
```

```

dfs(1);
printf("\n");
for(i=1;i<=n;i++)
{
    if(reach[i])
        count++;
}
if(count==n)
    printf("\n Graph is connected");
else
    printf("\n Graph is not connected");
}

```

Enter the number of vertices: 5

Enter the adjacency matrix

0 1 1 1 0

0 0 0 0 1

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

1->2

2->5

1->3

1->4

Graph is connected

## **PROGRAM 4:**

### **PROBLEM STATEMENT:**

**Implement and analyze topological sorting in a given directed graph**

### **CONCEPT & PROCEDURE**

Topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge  $uv$  from vertex  $u$  to vertex  $v$ ,  $u$

comes before  $v$  in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another;

A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). It uses stack data structure and DFS graph traversal method to generate topological ordering

#### **Source removal method:**

- Purely based on decrease & conquer
- Repeatedly identify in a remaining digraph a source, which is a vertex with no incoming edges
- Delete it along with all the edges outgoing from it.

The above algorithm is simply DFS with an extra stack. So time complexity is same as DFS which is  $O(V+E)$ .

```
#include<stdio.h>
void ts(int a[][], int n)
{
    int t[10], vis[10],
    stack[10], i, j, indeg[10], top=0, ele, k=0;
    top =0, ele, k=0;
    for(i=1; i<=n; i++)
    {
        t[i]=0;
        vis[i]=0;
        indeg[i]=0;
    }
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n; j++)
        {
            if(a[i][j]==1)
            {
                indeg[j]=indeg[j]+1;
            }
        }
    }
    printf("\n Indegree array :");
    for(i=1; i<=n; i++)
```

```

        printf("%d", indeg[i]);
    for(i=1; i<=n; i++)
    {
        if(indeg[i]==0)
        {
            stack[++top]=i;
            vis[i]=1;
        }
    }
    while(top>0)
    {
        ele = stack[top--];
        t[k++]=ele;
        for(j=1; j<=n; j++)
        {
            if(a[ele][j]==1 && vis[j]==0)
            {
                indeg[j]=indeg[j]-1;
                if(indeg[j]==0)
                {
                    stack[++top]=j;
                    vis[j]=1;
                }
            }
        }
    }
    printf("\n Topological Sorting \n");
    for(i=1; i<=n; i++)
        printf("%d  ", t[i]);
}

int main()
{
    int n, a[10][10], i, j;
    printf("Enter no. of vertices : ");
    scanf("%d", &n);
    printf("\n Enter Adjacency matrix : \n");
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n; j++)
        {

```

```

scanf("%d",a[i][j]);
    }
}
ts(a,n);
return 0;
}

```

Enter the number of nodes:

4

Enter the adjacent matrix

0 1 0 0

0 0 1 0

0 0 0 1

0 0 0 0

Indegree array is: 0 1 1 1

Topological ordering is:

1 2 3 4

## **PROGRAM 5:**

### **PROBLEM STATEMENT:**

Implement and analyze Kruskal's algorithm and find minimum cost spanning tree of a given connected undirected graph.

### **CONCEPT & PROCEDURE**

Kruskal's algorithm: Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes

every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

### **PROGRAM**

```
#include<stdio.h>
#include<stdlib.h>

int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);

int main()
{
    printf("\nEnter the no. of vertices:");
    scanf("%d",&n);
    printf("\nEnter the cost adjacency matrix:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&cost[i][j]);
            if(cost[i][j]==0)
                cost[i][j]=999;
        }
    }
    for(i=1;i<=n;i++)
    {
        parent[i]=0;
    }
    printf("The edges of Minimum Cost Spanning Tree are\n");
    while(ne < n)
    {
        for(i=1,min=999;i<=n;i++)
        {
            for(j=1;j <= n;j++)
            {
                if(cost[i][j] < min)
                {
                    min=cost[i][j];
                    a=u=i;
                    b=v=j;
                }
            }
        }
    }
}
```

```

        }
    }
    u=find(u);
    v=find(v);
    if(uni(u,v))
    {
        printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
        mincost +=min;
    }
    cost[a][b]=cost[b][a]=999;
}
printf("\n\tMinimum cost = %d\n",mincost);
}
int find(int i)
{
    while(parent[i])
        i=parent[i];
    return i;
}
int uni(int i,int j)
{
    if(i!=j)
    {
        parent[j]=i;
        return 1;
    }
    return 0;
}
}

```

### **OUTPUT:**

Enter the number of vertices

4

Enter the adjacency matrix

0 5 15 20

5 0 999 10

15 999 0 25

20 10 25 0

The edges of Minimum Cost Spanning Tree are

1 edge (1,2) =5

2 edge (2,4) =10

3 edge (1,3) =15

Minimum cost=30

## LAB PROGRAM 6:

### PROBLEM STATEMENT:

Implement and analyze Prim's algorithm and find minimum cost spanning tree of a given connected undirected graph.

### CONCEPT & PROCEDURE

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

The algorithm may informally be described as performing the following steps:

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

The time complexity of Prim's algorithm is  $O(V \log V + E \log V) = O(E \log V)$ ,

### PROGRAM

```
#include<stdio.h>
int main()
{
    int n, a[20][20], i, j, min, mincost, u, v, ne;
    int vis[20];
    printf("Enter the number of nodes : ");
    scanf("%d", &n);
    for(i=1; i<=n; i++)
        vis[i] = 0;
    printf("\n Enter the Cost Matrix or Adjacency Matrix : \n");
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n; j++)
        {
            scanf("%d", &a[i][j]);
            if(a[i][j] == 0)
            {
                a[i][j] = 999;
            }
        }
    }
}
```

```

    }
}
}
vis[1]=1;
ne=1;
mincost=0;
while(ne<n)
{
    for(i=1,min=999;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            if((a[i][j]<min) && (vis[i]!=0))
            {
                min=a[i][j];
                u=i;
                v=j;
            }
        }
    }
    if(vis[v]==0)
    {
        printf("Edge %d : (%d %d) cost %d", ne,u,v,a[u][v]);
        mincost=mincost+a[u][v];
        ne=ne+1;
        vis[v]=1;
        a[u][v] = a[v][u] = 999;
    }
}
printf("\n Minimum Cost = %d",mincost);
return 0;
}

```

### **OUTPUT:**

Enter the number of nodes

6

Enter the adjacency matrix

0	60	10	999	999	999
60	0	999	20	40	70
10	999	0	999	999	50
999	20	999	0	999	80

999	40	999	999	0	30
999	70	50	80	30	0

Spanning tree is shown below

Edge 1:(0 2) cost:10

Edge 2:(5 2) cost:50

Edge 3:( 4 5) cost:30

Edge 4:( 1 4) cost:40

Edge 5:(3 1) cost:20

Minimum cost=150

## LAB PROGRAM 7:

### PROBLEM STATEMENT:

Implement and analyze Dijkstra's algorithm to find the shortest path from a given source.

### CONCEPT & PROCEDURE

It is a greedy algorithm which finds the shortest path from the source vertex to all other vertices of the graph.

#### Steps

- Input a cost matrix for graph. Read the source vertex and n from user
- Create the array d [1...n] which stores the distance from source vertex to all other vertices of graph.
- Initialize distance to source vertex as 0(i.e. d [source] =0) and remaining vertices as 999.
- Create the array visited [1...n] which keeps track of all the visited nodes. Visit the source vertex and initialize visited [source] =1.
- For all adjacent vertices[vi,vi+1,...] for source vertex calculate distance using formula  $d[vi] = \min(d[vi], d[source] + \text{cost}[source][v1])$ . Update the array d[1...n].
- For all adjacent vertices find vertex vi which has minimum distance from source vertex.
- Initialize source = vi. Repeat the steps 4, 5 until there are no more vertices which are unvisited.
- Stop

Time complexity when Graph represented using adjacency matrix is  $O(V^2)$ . Graph represented using adjacency list can be reduced to  $O(E \log V)$  with the help of binary heap.

### PROGRAM

```
#include<stdio.h>
int main()
{
    int i,n, a[20][20], j, min, u,v, s[10], d[10],k;
    printf("Enter the number of nodes : ");
```

```

scanf("%d",&n);
printf("\n Enter adjacency matrix :\n");
for(i=1;i<=n;i++)
{
    for(j=1;j<=n;j++)
    {
        scanf("%d",a[i][j]);
    }
}
printf("\n Enter source vertex : \n");
scanf("%d",&v);
for(i=1;i<=n;i++)
{
    s[i]=0;
    d[i]=a[v][i];
}
d[v]=0;
s[v]=1;
for(k=2;k<=n;k++)
{
    min = 999;
    for(i=1;i<=n;i++)
    {
        if(d[i]<min && s[i]==0)
        {
            min = d[i];
            u=i;
        }
    }
    s[u]=1;
    for(i=1;i<=n;i++)
    {
        if(s[i]==0)
        {
            if(d[i]>d[u]+a[u][i])
            {
                d[i]=d[u]+a[u][i];
            }
        }
    }
}
}

```

```

        for(i=0;i<n;i++)
        {
            printf("\n Vertex %d distance %d ",v,d[i]);
        }
        return 0;
    }

```

Enter the number of vertices: 5

Enter the adjacency matrix

999 3 999 7 999

3    999 4    2 999

999 4    999 5    6

7    2 5    999 4

999 999 6    4 999

Enter the source vertex

1

The shortest distance from 1 is

1----->1=0

1----->2=3

1----->3=7

1----->4=5

1----->5=9

## LAB PROGRAM 8:

### PROBLEM STATEMENT:

Implement and analyze Floyd Warshall's algorithm to find the shortest path between all pairs of vertices in a given weighted connected graph.

### CONCEPT & PROCEDURE

Given a directed graph, find out if a vertex  $j$  is reachable from another vertex  $i$  for all vertex pairs  $(i, j)$  in the given graph. Here reachable mean that there is a path from vertex  $i$  to  $j$ . The reachability matrix is called transitive closure of a graph. The graph is given in the form of adjacency matrix say 'graph[V][V]' where graph[i][j] is 1 if there is an edge from vertex  $i$  to vertex  $j$  or  $i$  is equal to  $j$ , otherwise graph[i][j] is 0. If graph[i][j] is zero, then  $j$  is not reachable from  $i$ , otherwise  $j$  is reachable from  $i$ .

### **PROGRAM**

```
#include<stdio.h>

int min(int,int);

void floyds(int p[10][10],int n) {
    int i,j,k;
    for (k=1;k<=n;k++)
        for (i=1;i<=n;i++)
            for (j=1;j<=n;j++)
                if(i==j)
                    p[i][j]=0;
            else
                p[i][j]=min(p[i][j],p[i][k]+p[k][j]);
}

int min(int a,int b) {
    if(a<b)
        return(a);
    else
        return(b);
}
```

```
}
```

```
int main() {
```

```
    int p[10][10],n,i,j;
```

```
    printf("\n Enter the number of vertices:");
```

```
    scanf("%d",&n);
```

```
    printf("Enter the Matrix");
```

```
    for(i=1;i<=n;i++)
```

```
    {
```

```
        for(j=1;j<=n;j++)
```

```
        { scanf("%d", &p[i][j]);
```

```
        }
```

```
    }
```

```
    floyds(p,n);
```

```
    printf("\n Output Matrix :\n");
```

```
    for (i=1;i<=n;i++) {
```

```
        for (j=1;j<=n;j++)
```

```
            printf("%d \t",p[i][j]);
```

```
            printf("\n");
```

```
    }
```

```
    printf("\n The shortest paths are:\n");
```

```
    for (i=1;i<=n;i++)
```

```
        for (j=1;j<=n;j++) {
```

```
            if(i!=j)
```

```
                printf("\n <%d,%d->=%d",i,j,p[i][j]);
```

```
}  
}
```

### **OUTPUT:**

enter the no of vertices

3

Enter matrix

0 10 999

999 0 30

20 999 0

Output Matrix :

0 10 40

50 0 30

20 30 0

The shortest paths are:

<1,2>=10

<1,3>=40

<2,1>=50

<2,3>=30

<3,1>=20

<3,2>=30

### **LAB PROGRAM 9:**

#### **PROBLEM STATEMENT:**

Implement travelling salesman problem using dynamic programming.

#### **CONCEPT & PROCEDURE**

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

#### **Naive Solution:**

- 2) Consider city 1 as the starting and endpoint.
- 3) Generate all  $(n-1)!$  permutationsofcities.
- 4) Calculate cost of every permutation and keep track of minimum costpermutation.
- 5) Return the permutation with minimumcost.

Let the given set of vertices be  $\{1, 2, 3, 4, \dots, n\}$ . Let us consider 1 as starting and ending point of output. For every other vertex  $i$  (other than 1), we find the minimum cost path with 1 as the starting point,  $i$  as the ending point and all vertices appearing exactly once. Let the cost of this path be  $\text{cost}(i)$ , the cost of corresponding Cycle would be  $\text{cost}(i) + \text{dist}(i, 1)$  where  $\text{dist}(i, 1)$  is the distance from  $i$  to 1. Finally, we return the minimum of all  $[\text{cost}(i) + \text{dist}(i, 1)]$  values.

Dynamic programming algorithm can solve the problem in time  $O(n^2 \cdot 2^n)$  where  $n$  is the number of nodes in the graph

### **PROGRAM**

```
#include<stdio.h>

int a[10][10],visited[10],n,cost=0;

void mincost(int city)
{
    int i,ncity;
    visited[city]=1;

    printf("%d -->",city+1);

    ncity=least(city);

    if(ncity==999)
    {
        ncity=0;
        printf("%d",ncity+1);
        cost+=a[city][ncity];

        return;
    }
    mincost(ncity);
}
```

```

}

int least(int c)
{
    int i,nc=999;
    int min=999,kmin;
    for(i=0;i <n;i++)
    {
        if((a[c][i]!=0)&&(visited[i]==0))

            if(a[c][i] < min)
            {
                min=a[i][0]+a[c][i];

                kmin=a[c][i];
                nc=i;
            }
    }
    if(min!=999)
        cost+=kmin;

    return nc;
}

int main()
{
    int i,j;

    printf("Enter No. of Cities: ");

    scanf("%d",&n);

    printf("\nEnter Cost Matrix\n");

    for(i=0;i <n;i++)

```

```

    {
        for( j=0;j <n;j++)
            scanf("%d",&a[i][j]);

        visited[i]=0;
    }

    printf("\n\nThe Path is:\n\n");
    mincost(0);
    printf("\n\nMinimum cost:");

    printf("%d",cost);

}

```

OUTPUT:

Enter the no:of cities: 4

Enter the cost matrix

1 5 4 2

2 1 5 4

9 6 2 4

7 5 3 4

The path is:

1 → 4 → 3 → 2 → 1

Minimum cost:13

## LAB PROGRAM 10:

### PROBLEM STATEMENT:

Implement 0/1 Knapsack problem.

### CONCEPT & PROCEDURE

Given weights and values of  $n$  items, put these items in a knapsack of capacity  $W$  to get the maximum total value in the knapsack. In other words, given two integer arrays  $val[0..n-1]$  and  $wt[0..n-1]$  which represent values and weights associated with  $n$  items respectively. Also given an integer  $W$  which represents knapsack capacity, find out the maximum value subset of  $val[]$  such that sum of the weights of this subset is smaller than or equal to  $W$ . You cannot break an item, either pick the complete item or don't pick it (0-1 property).

Method1: Recursion.

Approach: A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

Optimal Sub-structure: To consider all subsets of items, there can be two cases for every item.

1. Case 1: The item is included in the optimal subset.
2. Case 2: The item is not included in the optimal set.
- 3.

Therefore, the maximum value that can be obtained from 'n' items is the max of the following two values.

1. Maximum value obtained by n-1 items and W weight (excluding nth item).
2. Value of nth item plus maximum value obtained by n-1 items and W minus the weight of the nth item (including nth item).

If the weight of 'nth' item is greater than 'W', then the nth item cannot be included and Case 1 is the only possibility.

## **PROGRAM**

```
# include<stdio.h>
void knapsack(int n, float weight[], float profit[], float capacity)
{
    float x[20], tp= 0;
    int i, j, u;
    u=capacity;

    for (i=0;i<n;i++)
        x[i]=0.0;

    for (i=0;i<n;i++)
    {
        if(weight[i]>u)
            break;
        else
        {
            x[i]=1.0;
            tp= tp+profit[i];
            u=u-weight[i];
        }
    }
}
```

```

    if(i<n)
        x[i]=u/weight[i];

    tp= tp + (x[i]*profit[i]);

    printf("\n The result vector is:- ");
    for(i=0;i<n;i++)
        printf("%0.2f\t\t\n",x[i]);

    printf("m Maximum profit is:- %0.2f", tp);

}

int main()
{
    float weight[20], profit[20], capacity;
    int n, i ,j;
    float ratio[20], temp;

    printf ("/n Enter the no. of objects:- ");
    scanf ("%d", &n);

    printf ("/n Enter the wts and profits of each object:- ");
    for (i=0; i<n; i++)
    {
        scanf("%f %f", &weight[i], &profit[i]);
    }

    printf ("/n enter the capacity of knapsack:- ");
    scanf ("%f", &capacity);

    for (i=0; i<n; i++)
    {
        ratio[i]=profit[i]/weight[i];
    }

    for(i=0; i<n; i++)
    {
        for(j=i+1;j< n; j++)

```

```

{
    if(ratio[i]<ratio[j])
    {
        temp= ratio[j];
        ratio[j]= ratio[i];
        ratio[i]= temp;

        temp= weight[j];
        weight[j]= weight[i];
        weight[i]= temp;

        temp= profit[j];
        profit[j]= profit[i];
        profit[i]= temp;
    }
}
}

knapsack(n, weight, profit, capacity);

}

```

**OUTPUT:**

Enter the no. of objects:- 3

Enter the wts and profits of each object:- 20 25 10

30 40 35

enter the capacity of knapsack:- 40

The result vector is:- 1.00

1.00

0.25

Maximum profit is:- 63.75

## **LAB PROGRAM 11:**

### **PROBLEM STATEMENT:**

Implement N Queens algorithm (Using backtracking).

### **CONCEPT & PROCEDURE**

Problem of placing N queens on N \* N chessboard in order that no two queens can attack each other which is known as 'n-Queens problem.

This problem contains three constraints: 1st, no two queens can share a same row. 2nd, no two queens can share a same column. 3rd, no two queens can share a same diagonal.

For the N-Queens problem, one way we can do this is given by the following:

- For each row, place a queen in the first valid position (row), and then move to the next row
- If there is no valid position, then one backtracks to the previous row and try the next position
- If one can successfully place a queen in the last row, then a solution is found. Now backtrack to find the next solution

$O(N!)$  is average case time complexity

### **PROGRAM**

```
#include<stdio.h>
```

```
#include<math.h>
```

```
int board[20],count;
```

```
int main()
```

```
{
```

```
int n,i,j;
```

```
void queen(int row,int n);
```

```
printf(" - N Queens Problem Using Backtracking -");  
printf("\n\nEnter number of Queens:");  
scanf("%d",&n);  
queen(1,n);  
return 0;  
}
```

```
//function for printing the solution
```

```
void print(int n)  
{  
    int i,j;  
    printf("\n\nSolution %d:\n\n",++count);
```

```
    for(i=1;i<=n;++i)  
        printf("\t%d",i);
```

```
    for(i=1;i<=n;++i)  
    {  
        printf("\n\n%d",i);  
        for(j=1;j<=n;++j) //for nxn board  
        {  
            if(board[i]==j)  
                printf("\tQ"); //queen at i,j position  
            else  
                printf("\t-"); //empty slot
```

```
}  
}  
}
```

```
/*function to check conflicts
```

```
If no conflict for desired position returns 1 otherwise returns 0*/
```

```
int place(int row,int column)
```

```
{
```

```
int i;
```

```
for(i=1;i<=row-1;++i)
```

```
{
```

```
//checking column and diagonal conflicts
```

```
if(board[i]==column)
```

```
return 0;
```

```
else
```

```
if(abs(board[i]-column)==abs(i-row))
```

```
return 0;
```

```
}
```

```
return 1; //no conflicts
```

```
}
```

```
//function to check for proper positioning of queen
```

```
void queen(int row,int n)
```

```
{
```

```

int column;

for(column=1;column<=n;++column)
{
    if(place(row,column))
    {
        board[row]=column; //no conflicts so place queen

        if(row==n) //dead end

            print(n); //printing the board configuration

        else //try queen with next position

            queen(row+1,n);
    }
}
}

```

### **OUTPUT:**

enter no of queens 4

Solution 1:

```

X Q X X
X X X Q
Q X X X
X X Q X

```

Solution 2:

```

X X Q X
Q X X X
X X X Q
X Q X X

```

## LAB PROGRAM 12:

### PROBLEM STATEMENT:

Implement sum of subset algorithm (Using backtracking).

### CONCEPT & PROCEDURE

Subset-Sum Problem is to find a subset of a given set  $S = \{s_1, s_2 \dots s_n\}$  of  $n$  positive integers whose sum is equal to a given positive integer  $d$ . It is assumed that the set's elements are sorted in increasing order. The state-space tree can then be constructed as a binary tree and applying backtracking algorithm, the solutions could be obtained.

Some instances of the problem may have no solutions

```
if(subset is satisfying the
    constraint)
    print the subset
    exclude the current element and consider next
    element
else
    generate the nodes of p
```

The running time is of order  $O(2^N)$ ,

### PROGRAM

```
#include<stdio.h>

#include<conio.h>

#define TRUE 1

#define FALSE 0

intinc[50],w[50],sum,n;

int promising(inti,intwt,int total){

    return(((wt+total)>=sum)&&((wt==sum) || (wt+w[i+1]<=sum))));

}

void main(){

    inti,j,n,temp,total=0;
```

```

printf("\n Enter how many numbers:\n");

scanf("%d",&n);

printf("\n Enter %d numbers to th set:\n",n);

for(i=0;i<n;i++){

    scanf("%d",&w[i]);

    total+=w[i];

}

printf("\n Input the sum value to create sub set:\n");

scanf("%d",&sum);

for(i=0;i<=n;i++)

for(j=0;j<n-1;j++)

if(w[j]>w[j+1]){

    temp=w[j];

    w[j]=w[j+1];

    w[j+1]=temp;

}

printf("\n The given %d numbers in ascending order:\n",n);

for(i=0;i<n;i++)

printf("%d \t",w[i]);

if((total<sum))

printf("\n Subset construction is not possible");else{

    for(i=0;i<n;i++)

    inc[i]=0;

    printf("\n The solution using backtracking is:\n");

```

```

        sumset(-1,0,total);
    }

}

voidsumset(inti,intwt,int total){
    int j;
    if(promising(i,wt,total)){
        if(wt==sum){
            printf("\n{\t");
            for(j=0;j<=i;j++)
                if(inc[j])
                    printf("%d\t",w[j]);
            printf("}\n");
        }else{
            inc[i+1]=TRUE;
            sumset(i+1,wt+w[i+1],total-w[i+1]);
            inc[i+1]=FALSE;
            sumset(i+1,wt,total-w[i+1]);
        }
    }
}
}

```

### **OUTPUT:**

enter the value of n

4

enter the set in increasing order

3

5

6

7

Enter maximum subset value of d

15

Solution is

3 5 7

### **LAB PROGRAM 13:**

#### **PROBLEM STATEMENT:**

Implement and compare simple string matching and KMP algorithms

#### **CONCEPT &PROCEDURE**

##### **Brute-Force StringMatching**

Align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until all m pairs match. But, if a mismatching pair is found, then the pattern is shift one position to the right and character comparisons are resumed. • The goal is to find i - the index of the leftmost character of the first matching substring in the text – such that  $t[i] = p[0], \dots, t[i+j] = p[j], \dots, t[i+m-1]$

$= p[m-1]$

Overall Time Complexity is  $O(m*n)$

##### **KMP String Matching**

The Knuth-Morris-Pratt (KMP) algorithm: It considers shifts in order from 1 to n-m, and determines if the pattern matches at that shift. Consider the situation when  $P[1 \dots 3]$  is successfully matched with  $T[1 \dots 3]$ . We then find a mismatch:  $P[4] \neq T[4]$ . Based on our knowledge that  $P[1 \dots 3] = T[1 \dots 3]$ , and ignoring symbols of the pattern and text after position 3, what can we deduce about where a potential match might be? In this case, the algorithm slides the pattern 2 positions to the right so that  $P[1]$  is lined up with  $T[3]$ . The next comparison is between  $P[2]$  and  $T[4]$ .

Overall Time Complexity is  $O(m + n)$

#### **PROGRAM**

```
#include<stdio.h>
```

```
#include<time.h>
```

```
#include<string.h>
```

```
Intlps[100];
```

```
void computeLPSArray(char pattern[])
```

```
{
    int len = 0, i;
    lps[0] = 0;
    i = 1;
    while(i < m)
    {
        if(pattern[i] == pattern[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else
        {
            if(len != 0 )
                len = lps[len-1];
            else
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

```
void kmp(char text[], char pattern[])
```

```
{
    int j=0, i=0, m, n;
    m = strlen(pattern);
    n = strlen(text);
    computeLPSArray(pattern);
    while(i < n)
    {
        if(pattern[j] == text[i])
        {
            j++;
            i++;
        }

        if (j == M)
        {
            printf("Found pattern at index %d \n", i-j);
        }
    }
}
```

```

        j = lps[j-1];
    }
    else if(pattern[j] != text[i])
    {
        if(j != 0)
            j = lps[j-1];
        else
            i = i+1;
    }
}
}

```

```

Void bruteforce(char text[],char pattern[])
{
    inti,j,m,n;
    m=strlen(text);
    n=strlen(pattern);

    for (i = 0; i<= m - n; i++)
    {
        for (j = 0; j <n; j++)
        {
            if (text[i+j] != pattern[j])
                break;
        }
        if (j == n)

    printf("Using Brute force Match found at index %d",i));
        return;
    }
    printf("Using Brute: No Match found");
}

```

```

Intmain()
{
    char text[50],pattern[50];
    clock_t st1,st2,end1,end2;
    printf("Enter the text\n");
}

```

```

gets(text);
printf("Enter the pattern\n");
gets(pattern);
st1=clock();
bruteforce(text,pattern);
end1=clock();
printf("Time required for bruteforce match %f",(end1-st1)/CLOCKS_PER_SEC);
st2=clock();
kmp(text,pattern);
end2=clock();
printf("Time required for kmp match %f",(end2-st2)/CLOCKS_PER_SEC);
}

```

### **OUTPUT:**

Enter the text: Hello Good Morning

Enter the pattern: Good

Time required for bruteforce match:0.00423000

Time required for kmp match: 0.00135000

### **LAB PROGRAM 14:**

#### **PROBLEM STATEMENT:**

Implement prefix computation algorithm by using multiple threads or processors

#### **CONCEPT & PROCEDURE**

prefix sum takes an associated binary operator  $\oplus$  and an ordered set  $[a_1, \dots, a_n]$  of  $n$  elements and returns the ordered set  $[a_1, (a_1 \oplus a_2), \dots, (a_1 \oplus a_2 \oplus \dots \oplus a_n)]$ .

The prefix sum or simply scan of a sequence of numbers  $x_0, x_1, x_2, \dots$  is a second sequence of numbers  $y_0, y_1, y_2, \dots$ , The sums of prefixes (running totals) of the input sequence:

$y_0 = x_0$   
 $y_1 = x_0 + x_1$   
 $y_2 = x_0 + x_1 + x_2$

...

For instance, the prefix sums of the natural numbers are the triangular numbers:

Input numbers	1	2	3	4	5	6	...
Prefix values	1	3	6	10	15	21	...

### **PROGRAM**

```
#include<stdio.h>
#include<omp.h>
int main()
{
    int arr[50], n, i, tid;

    int prefixSum[50];
    printf("Enter the number of elements \n");
    scanf("%d",&n);
    printf("Enter the elements into the array : \n");
    for(i=0;i<n;i++)
        scanf("%d",&arr[i]);
    prefixSum[0]=arr[0];
    omp_set_num_threads(2);
#pragma omp parallel
    {
        #pragma omp for private(tid)

        for(i=0;i<n;i++)
        {
            tid = omp_get_thread_num();
            printf("Thread is %d ",tid);
            prefixSum[i] = prefixSum[i-1] + arr[i];
        }
    }

    printf("\n The prefix sum is : ");
    for(i=0;i<n;i++)
    {
        printf("%d \n",prefixSum[i]);
    }
}
```

```
    }  
    return 0;  
}
```

### Compilation

```
cc -fopenmp pgmname.c
```

### OUTPUT:

Enter the no.of elements : 5

Enter the elements into the array :

10 20 30 50 40

Thread is 0 Thread is 0 Thread is 0 Thread is 1 Thread is 1

The prefix sum is : 10

30

60

110

150

